Outline:

1. Introduction
2. Overview
3. Updated UML
4. Design
5. Resilience to Change
6. Answer to Questions
7. Extra Credit Features
8. Final Questions
9. Conclusion

Introduction

This documentation describes the architecture and design for the Game of Quadris, which is a Latinization of the game Tetris.

Overview

Class Introduction

Depending on the needs of the Project, the major part of the program is separated into several classes.

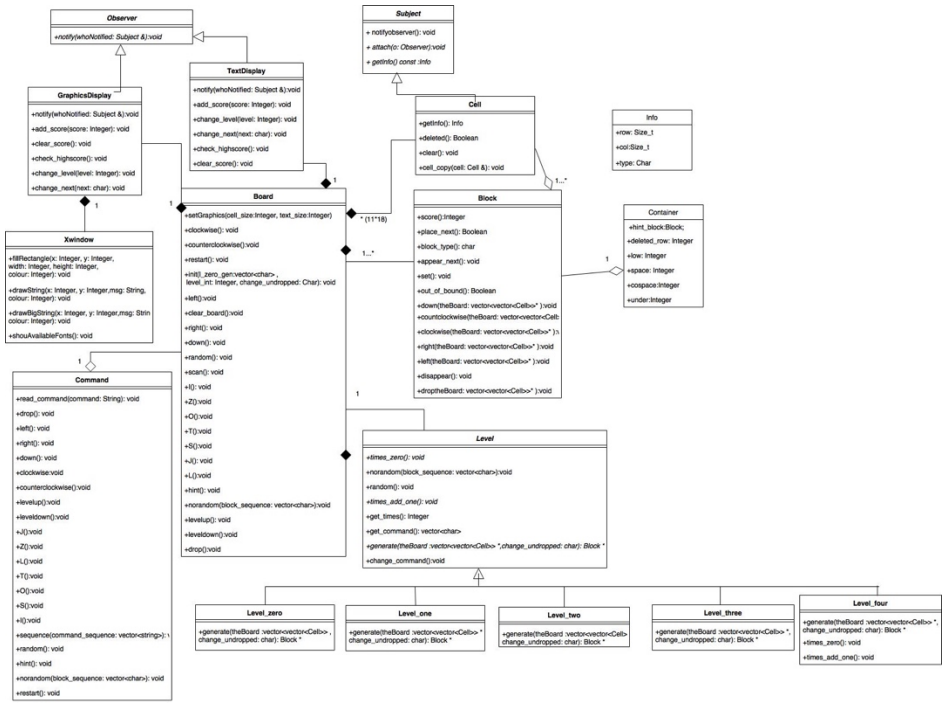
The main classes are as follows:

- Board
    - Board class is a class which contains the actual board of the game
    - The actual board is saved as a 2D vector of Cells
    - Board class is responsible for storing the block that user is controlling and the next block
    - Board class keeps track of all the blocks that have been dropped in order give bonus score when a block completely disappears
    - Board class also stores the current level
    - Board class' main responsibility is keeping the game state
- Block
    - Block class is a class which keeps track of a set of Cells that makes up a block
    - It keeps track of the cells it has in a vector of Cell pointers

        - Block class stores its type which can be one of ‘I’, ‘J’, ‘L’, ‘O’, ‘S’, ‘Z’, ‘T’, or ‘*’
        - A Block stores the level it is generated from
        - A Block has an integer field which tells the number of Cell it has that is still on the board
    - Cell
        - Cell class is a class which represents a single cell or position on the Board
        - A cell store its position as two unsigned integer (row, column)
        - A Cell could be any of the block type or nothing at one time
        - A Cell also has a Boolean field (landed) which tells whether it is occupied or not
    - Level
        - Level class is an abstract superclass of concrete levels of the game
        - Level dictates the policy for selecting the next block by providing a generate method which constructs a Block
        - Level class contains a sequence of predetermined Blocks which is used for Level 0 and the norandom option for the game, in a vector of Char
    - Level_zero
        - Level_zero is a concrete subclass of superclass Level, and it represents Level 0 of the game
        - Level_zero generates Blocks in same order as the sequence which is stored in the vector of Char
    - Level_one
        - Level_one is a concrete subclass of superclass Level, and it represents Level 1 of the game
        - Level_one randomly generates Block with 1/12 probability for type ‘S’ and ‘Z’ and 1/6 probability for rest of the Block types
    - Level_two
        - Level_two is a concrete subclass of superclass Level, and it represents Level 2 of the game
        - Level_two randomly generates Block with equal probability for all types
    - Level_three
        - Level_three is a concrete subclass of superclass Level, and it represents Level 3 of the game
        - Level_three randomly generates Block with 2/9 probability for type ‘S’ and ‘Z’ and 1/9 probability for rest of the Block types
        - Block generated by Level_three are heavy, which means they move down by one Cell unit when rotated, or moved

- Level_four
    - Level_four is a concrete subclass of superclass Level, and it represents Level 4 of the game
    - Level_four randomly generates Block same probability as Level_three
    - Block generated by Level_four are heavy
    - When Level_four has generated 5 blocks before the player clearing one row a Block of type '*' is dropped in the center of the Board
- Command
    - Command class is a class which handles player's command
    - Command serves as the "controller" of the game
- TextDisplay
    - TextDisplay class is a class which provides a text-based display for the Game Board
    - It is responsible for the viewing of the game
- GraphicsDisplay
    - GraphicsDisplay class is a class which provides a text-based display for the Game Board
    - It is responsible for the viewing of the game

Updated UML



Design

The system employs good object-oriented design. In total, the program has 30 files(.cc and .h) and 9 main classes. Each class is responsible for just one function. In some ways this achieves high cohesion and low coupling, and also enhances the flexibility to accommodate the possibility of introducing new features into the system with minimum changes.

Heap Memory Control

In order to achieve 0 memory leak during the execution of our game, we applied unique_ptr on all heap allocated object. Every object that was created on the heap was wrapped in a unique_ptr object (TextDisplay, GraphicsDisplay in Board etc.). Since every unique_ptr will call the dtor of contained object pointer when going out of scope, it ensures every piece of heap memory can be freed when the problem is ended.

MVC Pattern

We applied MVC Pattern on our overall design of the game. MVC divides the system in three basic components and assigns the basic functions, the presentation of the data, the state of the data and the control of the data to each part. The Board class is used to encapsulate the data as Model, which stores the game's state and provides methods that allows manipulation and management of data it stores. It does not rely on View and Controller. TextDisplay and GraphicsDisplay is desgiend for text view and graphics view, displaying a single Model, which eliminates code duplication. Command Class plays the role as the Controller, which interprets the player's input and manipulates the data stored in Model (Board).

Factory Method Pattern (Level)

The class Level is designed as a part of the Factory Method Pattern to generate Blocks. First an abstract superclass Level is created and provides a pure virtual function generate(), that its subclasses inherits and implement on their own. Each subclass of the Level class implements generate() method according to the specific properties of that level (e.g. level 3 and 4 generates block with gravity) according to specification of the level. This design ensures low coupling, because if we need to add new properties to a certain level we can simply adjust in the concrete Level subclass (Level_zero etc.), without changing where generate() method is called. Which generate is called is determined during runtime (dynamic dispatch), and the caller does not have to know the exact object it is called upon, which saves lots of code. In addition, if we want to add

additional level we can easily create a concrete new class inheriting the abstract Level class and override the generate() method.

Observer Pattern

We used Observer Pattern to ensure that both TextDisplay and GraphicsDisplaying tracks every change of the Cells and displays them accurately. Every Cell in the Board can be in different state (eg. ‘L’, ‘Z’, ‘O’ etc. each represented by a different colour in GraphicsDisplay), and when a Cell’s type is changed the two display will be notified immediately. Once notified both display will check the info of the cell and triggering changes on the display do accurately depict the game.

Resilience to Change

In our implementation of the game, we have partitioned tasks in different objects and reduced coupling as much as possible between them. All properties of object are implemented in each object class, the level’s generation of blocks by factory pattern. User command is interpreted in the Command class and it controls/manipulates the game state stored in Board. By doing this, we only need to have very small changes to add new components. The details of expected variation to possible given changes are discussed below.

Case 1: Change in Rules

If some rule changes, only classes that are related to the change should be changed. If the Block shape or movement changes, only constructor or the methods of the block should be changed. If generation rules of Blocks for a level is changed only the overridden generate method of the concrete subclass of Level affected should be changed. If the score calculation is changed, only the Board needs to be changed, both displays simply receives the score and displays them. If there are other changes in the rules other than the changes mentioned above, these changes can be implemented in the Grid class. Thus, only main, Board and few other classes will be recompiled when rules change.

Case 2: Change in input syntax

When input syntax changes, only the Command class should be changed. Since input is interpreted entirely in the Command class. Therefore the only file that requires recompilation would be the Command class (command.cc).

Case 3: New Features

To introduce new features, new class should be created and few other classes may need to be changed. For example, adding new level needs to create a new level class

inherited from Level. Main should be changed to include the added level class. Therefore, only main and the added Level should be compiled. If we want to make it a two player game, we can initialize two Boards in main and add field in the board to distinguish between the two. We also need to modify TextDisplay and GraphicsDisplay to display 2 boards and change notify function to distinguish which board is being changed. Thus, main, Board and the two display classes will be recompiled for this feature.

Answer to Questions

Question: How could you design your system (or modify your existing design) to allow for some
generated blocks to disappear from the screen if not cleared before 10 more blocks have fallen?
Could the generation of such blocks be easily confined to more advanced levels?

To allow for some generated blocks to disappear from the screen if not cleared before 10 more blocks have fallen, we could add a two fields in the block class: one bool field in the block class to indicate whether the block has such disappearing property; another integer field that is default to 10. When each block has landed a method “scan” will be called (originally only checks if any row is full) to go through the vector of blocks (that have not been fully cleared) which is already in our design as a field in the board class, to first decreases the integer field by one and then check if a block has such property and if the integer field is zero. If it has such property and the integer field is zero, then this block disappears by manipulating the cells type through cell pointers within the block.
Generation of such block can be easily confined to more advanced level as all level 1-4 are subclasses of a level class, and generation of blocks is through virtual method of the level class. We can easily edit the generation method of more advanced level to accommodate this change.

Question: How could you design your program to accommodate the possibility of introducing
additional levels into the system, with minimum recompilation?

We have created a base class called Level, which has a public virtual method called generate, and we make several subclasses inherited from Level, in these subclasses we override the method generate such that we have different ways to generate blocks.

If we need to add additional level, we just need to add a new subclass inherited from Level and give a different way to generate. In main.cc, including level.h and new_level.h is enough. When compiling only main.cc and new_level.cc will be compiled.

Question: How could you design your system to accommodate the addition of new command names, or changes to existing command names, with minimal changes to source and minimal recompilation? (We acknowledge, of course, that adding a new command probably means adding a new feature, which can mean adding a non-trivial amount of code.) How difficult would it be to adapt your system to support a command whereby a user could rename existing commands (e.g. something like rename counterclockwise cc)? How might you support a macro" language, which would allow you to give a name to a sequence of commands? Keep in mind the effect that all of these features would have on the available shortcuts for existing command names.

Revising our original answer, we have decided to add a Command class to interpret player's command to more closely abide to the MVC Pattern. To add additional command names or change command names we could simply add methods with in the Command class. When compiling only the Command class and classes that are affected by the new features will be recompiled, ensuring minimal recompilation.
To support user renaming of existing commands we can set command names as string fields with default value in the Command class and when users enters "rename" followed by the old command name then the new command name; a rename method is then called which changes an old command names to the new ones by reassigning read string value to the variables. These variables will be used in read_command method to compare with the input, so when these variables are changed entering the newly defined name will executes the command.
To support a macro language that allows user to give a name to a sequence of commands, similarly we could add a map field in the Command class where the key will be the name of user defined name in string and the value would be the sequence of command associated with that name also in string. Whenever a command is read, the read input method will check if the command is a key, and then the value is put in a stream. The method then will read through this stream executing the corresponding command by calling relating methods.

Extra Credit Features

Memory management via vector smart pointers

We applied unique_ptr to prevent memory leak. For how they have been implemented, please check Design part of this report. One challenge during the implementation is distinguishing when to use raw pointer to create a has relationship and when to take ownership with unique_ptr. A Block has a field of vector of cell pointers to form a “has a” relationship. When the Block is destroyed no Cells should ever be destroyed from the Board. Thus raw pointers need to be used for this purpose. Another challenge during the implementation is to accommodate the constantly changing current and next block. We have to utilize methods such as “reset” and “move” to transfer property ownership and facilitate calls to dtors.

Next block when level up and level down

As required by the guideline, leveling up or down the block showing as next still comes next but subsequent blocks are generated using the new level. The slight issue with this is that after changing the level, the player has to do two drop to start experiencing the features of that level, which creates a feeling of lag. So we have decided to implement a feature that keeps the next block’s type while giving the next block properties of the next level when leveling up or down is called to enhance player experience. The challenge of implementing this feature is to be able to create any block type we desire for any level generator. So we added a parameter in the generate method in the abstract super Level class and implemented this for all Level subclasses which inherits this method. When a ‘N’ is given to the parameter, the generate method will return a block according to the generation specification of that level, otherwise when a Block type is given, it will generate a specific Block of that type which also has the properties of that level.

Final Question

1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

   The most important thing we have learned is that in order to adopt good design and have high resilience to change, it is essential to come up with the basic structure of the game before writing any code or implementation. Through our experience, we find that it is much easier to formulate the design first instead of implementing first and try to modify the whole structure throughout the process. We also find that it is much more convenient to debug and modify our code by utilizing proper design patterns whatever we see viable. In addition, a lesson we learned that it is sufficient to initialize the random seed once, instead of initializing the seed every time rand is called, which would have caused biased random number being generated. Another important lesson we’ve learned is that

communication is essential. If one of us came across any difficulties that could not be solved alone in a reasonable amount of time, we should try to seek assistance from teammates, TAs or professor. Also, it is important to delegate tasks reasonably, so that the burden of the assignment could be could be shared while ensuring the assignment could be finished with every team being fully involved. Lastly, it is of great importance that each of the team member complete the assigned tasks on time so that the assignment could progress as a whole.

2. What would you have done differently if you had the chance to start over?

   If we can start over, we will reuse the method when possible, which could reduce the workload of writing code for same functions. In addition, we will maximum the independence between each part in order to achieve higher cohesion and lower coupling to reduce the compilation dependency and improve the efficiency of debugging. Also, we would make good use of Template functions which we learned during the last lecture, to reduce code used for movement and rotation significantly. Lastly we should consider some special cases before we start the project as well, because modifying a small part in the whole project sometimes leads to modification of many parts (return type of functions, const or non-const methods, parameters/default parameters…). To conclude, even though it is a large and hard project, we successfully finish it in time and added some extra bonus features, and we are proud to present it before everyone.

Conclusion

The Game of Quadris is successfully completed by our joint effort. We learned how design a program with teammates, enhanced our programming skills and are more familiar with object-oriented design. Moreover, we feel very lucky to have each other as our teammate, we all expect the next cooperation.